

# Go Static: Contextualized Logging Statement Generation

---

Yichen Li<sup>1</sup>, Yintong Huo<sup>1</sup>, Renyi Zhong<sup>1</sup>, Zhihan Jiang<sup>1</sup>,  
Jinyang Liu<sup>1</sup>, Junjie Huang<sup>1</sup>, Jiazhen Gu<sup>1</sup>, Pinjia He<sup>1</sup>,  
Michael R.Lyu<sup>1</sup>

<sup>1</sup>Chinese University of Hong Kong

*Proceedings of the ACM on Software Engineering, 1(FSE2024)*

# 目次

---

研究背景

既存手法の限界

問題定義

提案手法：SCLogger

実験評価

まとめと今後の課題

所感

# ログとは

---

## システムが記録する動作履歴をログという

- ログは、デバッガが使えないような本番環境での不具合原因特定に有用[1].
- システムが動作を記録することそのものをロギングという

[1] Yuan, D., Park, S., & Zhou, Y. (2012). Characterizing logging practices in open-source software. 2012 34th International Conference on Software Engineering (ICSE), 102-112.

# ログステートメント

プログラムの実行中にロギングを行うために、ソースコード中に記述する命令文をログステートメントという

- JavaのAPIであるLog4jやSlf4j, Pythonの標準モジュールであるloggingなどがログステートメントを提供している

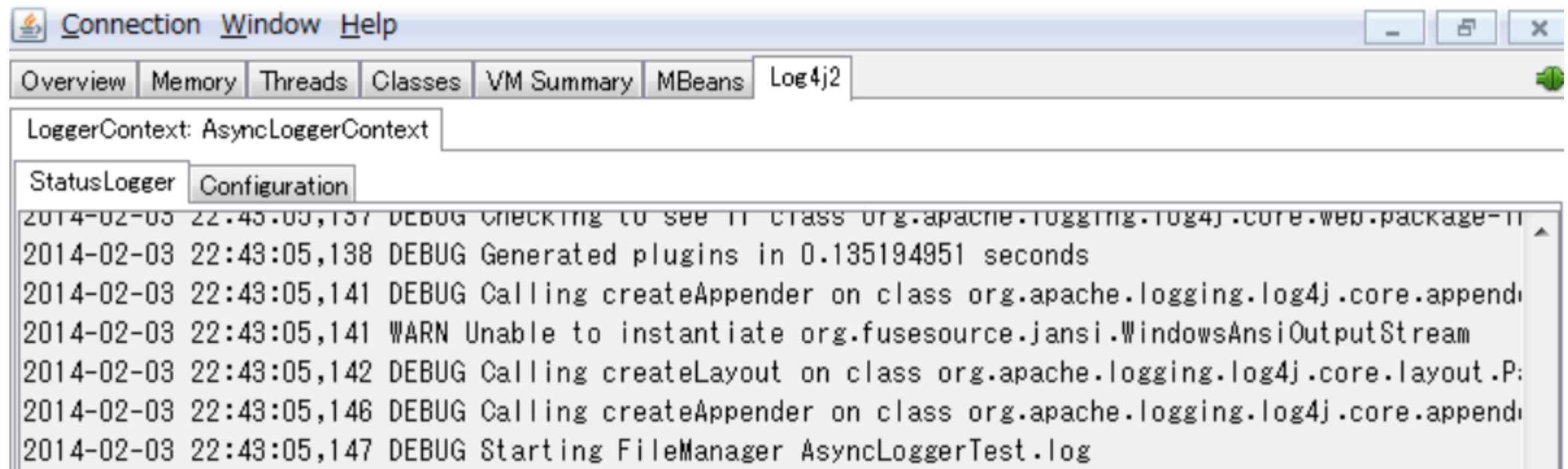


図 1 : Log4jのログステートメントにより出力されるログの例

# ログステートメントとログの対応関係(Log4j)

---

```
logger.debug("Debugging user: username=" + username + ", id=" + userId);
```

Log4jのログステートメント



```
2025-05-26 10:30:00 [DEBUG] Main - Debugging user: username=vanilla, id=42
```

上記のログステートメントにより出力されるログ

# ログステートメントとログの対応関係(Log4j)

```
logger.debug("Debugging user: username=" + username + ", id=" + userId);
```

Log4jのログステートメント

```
2025-05-26 10:30:00 [DEBUG] Main - Debugging user: username=vanilla, id=42
```

上記のログステートメントにより出力されるログ

※usernameとuserIdは変数

# ログステートメントとログの対応関係(Log4j)

ログステートメントの構成要素として、  
ログレベル、ログ変数、ログテキスト  
の3つが挙げられる

```
logger.debug("Debugging user: username=" + username + ", id=" + userId);
```

Log4jのログステートメント

ログレベル  
(そのログの重要度  
を示す)

ログテキスト

```
2025-05-26 10:30:00 [DEBUG] Main - Debugging user: username=vanilla, id=42
```

上記のログステートメントにより出力されるログ

※usernameとuserIdは ログ変数

# 自動ログステートメント生成の必要性

---

## 手動でのログステートメント記述の課題

- 開発者の経験に依存
- プロジェクト内での一貫性不足
- 適切な場所・内容の判断が困難

## 自動化による利点

- 開発効率の向上
- 一貫したログスタイル
- 人的エラーの削減

# 既存の自動ログ生成手法

---

最新のロギングステートメント生成アプローチとして、  
LANCEが挙げられる

- GPT3.5等のTransformerモデルを使用したフレームワーク
- 入力：ログステートメントを生成したいJavaメソッド
- 出力：ログステートメントそのもの、挿入する箇所(行)
- 事前訓練データセット：1,465のJavaプロジェクトから抽出した約690万のJavaメソッドに加工を施したもの

# 既存手法の3つの限界

## 限定的で不十分な静的スコープ：

- メソッドの実行順等，メソッドの外にある重要な情報を見逃している
  - 既存手法はメソッドの内部しか見ていない

## 一貫性のないログスタイル：

- ログレベル決定のための基準，ログテキストのフォーマット等が，生成前からソースコード中にあったログステートメントと異なってしまう

## 変数型情報の欠如：

- 変数の役割が把握できなく，ログテキストで変数を誤用してバグやエラーを出してしまう
  - 例えば，int型なら必ずstr型に変換しないとコンパイルエラーが出る

# 研究目的とアプローチ

---

**研究目的:** メソッドレベルを超えた文脈情報を活用して、より効果的なログステートメント生成を実現する

**アプローチ:**

- 以下の事項を考慮したログステートメント生成手法  
SCLoggerの提案
  - 静的解析による文脈情報の拡張
  - プロジェクト固有のログスタイル適応
  - 変数型情報の活用

# SCLoggerの概要

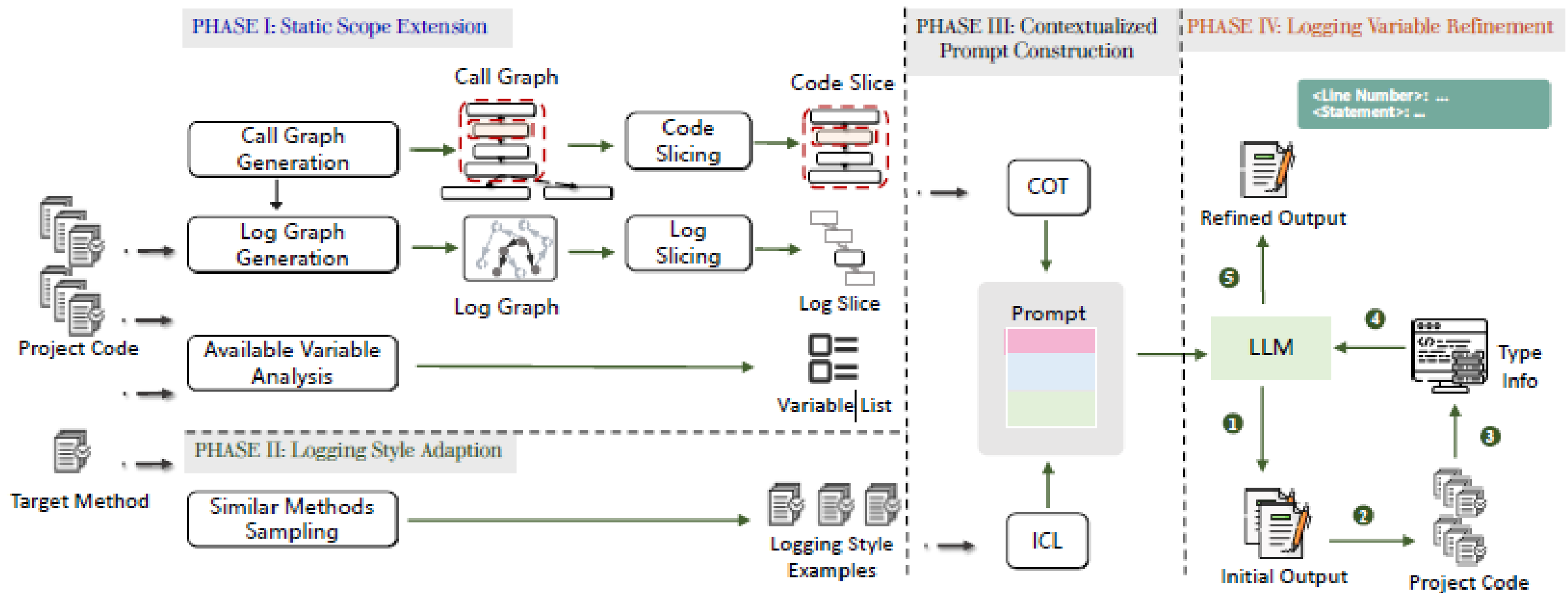


Fig. 4. The overview workflow of SCLogger.

入力：ログステートメントを生成したいJavaメソッドとその文脈情報

出力：ログステートメントそのもの，挿入する箇所(行)

# Phase I: 静的スコープ拡張

---

## 3つの情報を抽出：

- 2呼び出し以内の呼び出し関係
- 実行順序における前後のログ
- そのメソッドが利用可能なすべてのパラメータ、ローカル変数、クラス変数、継承変数

## 技術的詳細：

- Context-sensitive pointer analysis[2]により構築された精密な呼び出しグラフ

[2]Li, Y., Tan, T., Møller, A., & Smaragdakis, Y. (2018). Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 1-29.

# Phase II: ログスタイル適応

In-Context Learning (ICL) 戦略を採用

- 与えられた文脈から新しいタスクを理解

BM25類似度関数[3]を用いて例となる類似メソッドを、プロジェクト中の既存のメソッドから選択

- 各キーワードの頻度を元に、メソッド同士の類似度を決定

トップ5の類似メソッドを選択

選択した類似メソッドの中のログをFew-shot promptの一部として、この後のプロンプトに組み込む

[3]Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. Foundations and TrendsR in Information Retrieval 3, 4 (2009), 333–389.

# Phase III: 文脈化プロンプト構築

---

Chain-of-Thought (CoT) 戦略を採用

静的解析情報をステップ形式で整理：

- 呼び出し関係の説明
- 前後のログ情報
- 利用可能変数リスト
- 類似メソッド例の提示

ICLとCoTを組み合わせた文脈化プロンプト

# プロンプト例

## Example Prompt

### Instruction:

Insert a logging statement to the following code using <API>.  
With the format <Line Number>: # <Statement>: #

### Java Code:

```
public void stateChanged(Service service) {  
...  
}
```

Let's think step by step. First, the given method was called  
by <method A>, the code of called <methodB> is..

Second, the succeeding and proceeding logs are: <log>  
<log>...

Third, available variables of this method are <var>, <var>..

Finally, keep consistent log style within current project, here  
are logging examples from current project:

<example>:<label>

<example>:<label>

...

} ログステートメントを生成する  
メソッド

} Phase1で集めた文脈情報

} Phase2で集めたプロジェクト内の  
類似メソッドにおける  
ログステートメントの例

# Phase IV: ログ変数の改良

## Example Prompt

### Instruction:

Insert a logging statement to the following code using <API>.  
With the format <Line Number>: # <Statement>: #

### Java Code:

```
public void stateChanged(Service service) {  
...  
}
```

Let's think step by step. First, the given method was called by <method A>, the code of called <method B> is..  
Second, the succeeding and proceeding logs are: <log>  
<log>..  
Third, available variables of this method are <var>, <var>..  
Finally, keep consistent log style within current project, here are logging examples from current project:  
<example>:<label>  
<example>:<label>  
...

## Refined Output

```
<Line Number>: 2  
<Statement>: log.info("State Changed to  
" + service.getServiceState() + "for" +  
service.getName());
```

Send

1

LLM

5

Refine

4

2

```
<Line Number>: 2  
<Statement>: log.info(" State Changed to  
service.State()");
```

## Initial Output

Please rethink the variable usage with detailed type information about <Var>

```
public interface Service extends Closeable {  
public enum STATE {  
...  
}  
STATE getServiceState();  
String getName();  
}
```

3  
Get Detailed Type  
Information of <var>

## 2段階の変数型改良メカニズム：

1. LLMに推論でログ変数を特定させる
2. 選択された変数の詳細を静的解析で取得
3. LLMに取得型情報を用いて変数使用を再考・修正させる

# 実験設定

---

## 対象プロジェクト: 10のオープンソースJavaプロジェクト

- ActiveMQ, Hadoop, HBase等
- 330K-2.12M SLOC
- 1,978-15,744 ログステートメント

## ベースライン:

- LANCE/LANCE2.0 (専用手法)
- GPT-3.5, GPT-4, Llama-2-70b等のLLM

## 評価指標:

- ログステートメントの位置: Position Accuracy (PA)
- ログステートメントの内容: Level Accuracy, Variable Precision/Recall, BLEU/ROUGE

# RQ1：SCLOGGERは既存のアプローチと比較してどの程度効果的か？

Table 2. Logging statements generation results from both *where-to-log* and *what-to-log* dimensions.

| Model       | Posistion | Logging Levels |       | Logging Variables |        |       | Logging Texts |        |         |         |
|-------------|-----------|----------------|-------|-------------------|--------|-------|---------------|--------|---------|---------|
|             | PA        | L-ACC          | AOD   | Precision         | Recall | F1    | BLEU-1        | BLEU-4 | ROUGE-1 | ROUGE-L |
| LANCE       | 0.501     | 0.574          | 0.763 | 0.657             | 0.414  | 0.508 | 0.207         | 0.110  | 0.179   | 0.175   |
| LANCE2.0    | 0.563     | 0.601          | 0.807 | 0.632             | 0.508  | 0.563 | 0.219         | 0.113  | 0.275   | 0.266   |
| Davinci-003 | 0.307     | 0.470          | 0.714 | 0.626             | 0.544  | 0.582 | 0.267         | 0.128  | 0.288   | 0.295   |
| Llama-2-70b | 0.248     | 0.442          | 0.682 | 0.506             | 0.477  | 0.490 | 0.209         | 0.070  | 0.218   | 0.219   |
| GPT-3.5     | 0.395     | 0.495          | 0.727 | 0.618             | 0.496  | 0.550 | 0.164         | 0.064  | 0.176   | 0.174   |
| GPT-4       | 0.518     | 0.562          | 0.779 | 0.634             | 0.611  | 0.622 | 0.285         | 0.138  | 0.317   | 0.321   |
| SCLOGGER    | 0.612     | 0.794          | 0.914 | 0.758             | 0.735  | 0.746 | 0.493         | 0.329  | 0.517   | 0.509   |

## 大幅な性能向上:

- Position Accuracy(位置精度): 8.7%向上（対LANCE2.0）
- Level Accuracy(ログレベル精度): 32.1%向上
- Variable Precision(ログ変数精度): 19.6%向上
- BLEU-4 Score(人間のものとどれだけ似てるか): 138.4%向上

# RQ2：SCLOGGERの各フェーズが与える影響は？

Table 3. Ablation Study of SCLOGGER.

| Ablation                        | Posistion | Logging Levels |       | Logging Variables |        |       | Logging Texts |        |         |         |
|---------------------------------|-----------|----------------|-------|-------------------|--------|-------|---------------|--------|---------|---------|
|                                 | PA        | L-ACC          | AOD   | Precision         | Recall | F1    | BLEU-1        | BLEU-4 | ROUGE-1 | ROUGE-L |
| SCLOGGER                        | 0.612     | 0.794          | 0.914 | 0.758             | 0.735  | 0.746 | 0.493         | 0.329  | 0.517   | 0.509   |
| w/o Loging Scope Extension      | 0.579     | 0.702          | 0.858 | 0.720             | 0.711  | 0.716 | 0.430         | 0.278  | 0.468   | 0.469   |
| w/o Logging Style Adaption      | 0.549     | 0.679          | 0.869 | 0.752             | 0.696  | 0.723 | 0.354         | 0.191  | 0.393   | 0.386   |
| w/o Logging Variable Refinement | 0.614     | 0.791          | 0.912 | 0.708             | 0.654  | 0.680 | 0.483         | 0.348  | 0.507   | 0.503   |

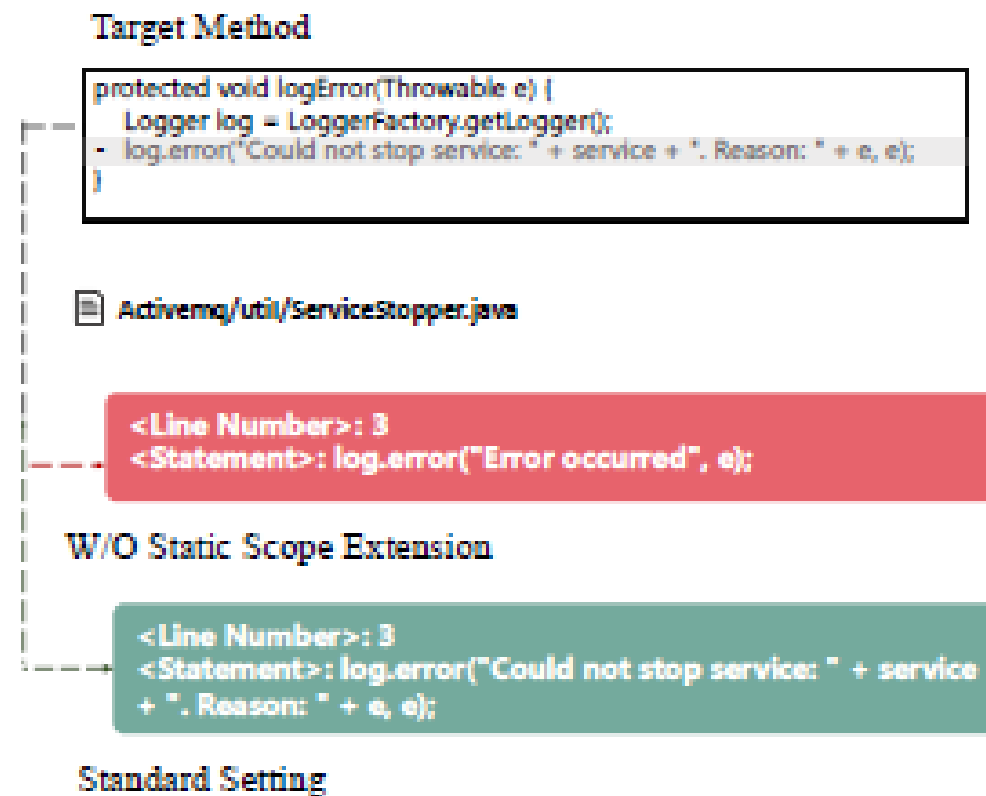
各フェーズの性能への影響を評価

静的スコープ拡張なし: PA -5.4%, L-ACC -11.6%

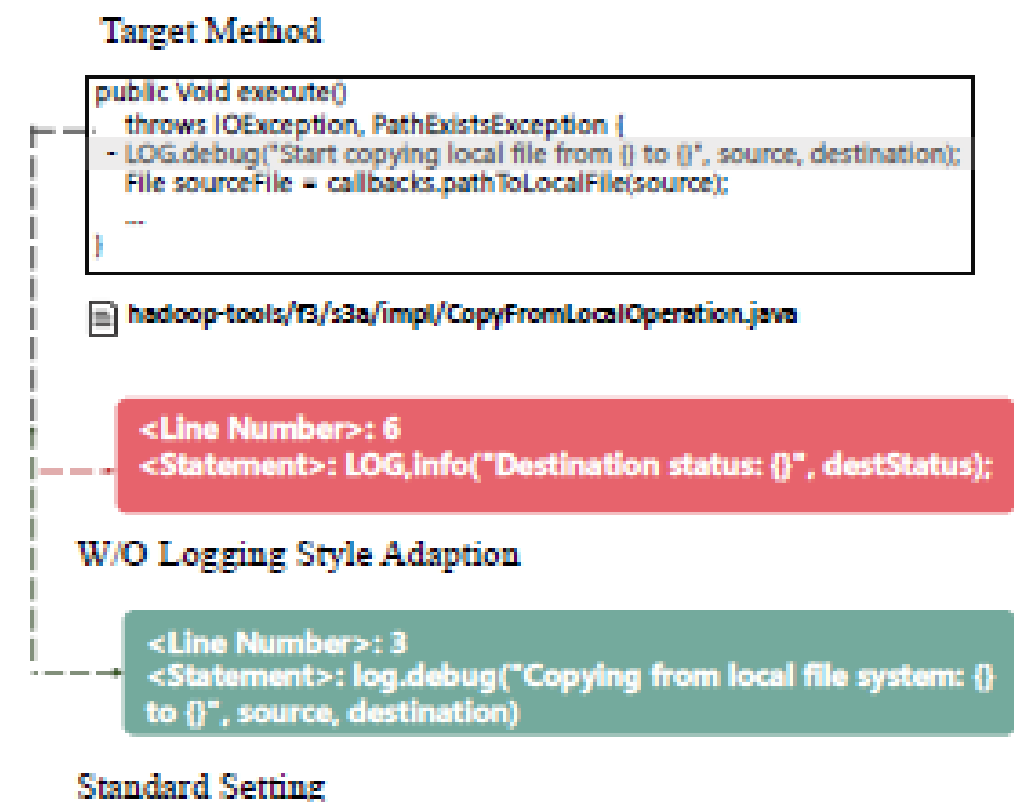
ログスタイル適応なし: PA -10.3%, BLEU-4 -41.9%

変数精緻化なし: Variable F1 -8.8%

# RQ3：SCLOGGERは異なるバックボーンモデルに対してどの程度汎用性があるか？



(a) Removing the phase of static scope extension.



(b) Removing the phase of logging style adaption.

具体例による各段階の効果実証

静的スコープ拡張により適切なエラーログを生成

ログスタイル適応により一貫したパターンを学習

# RQ4：異なるログステートメント例の取り方が与える影響は？

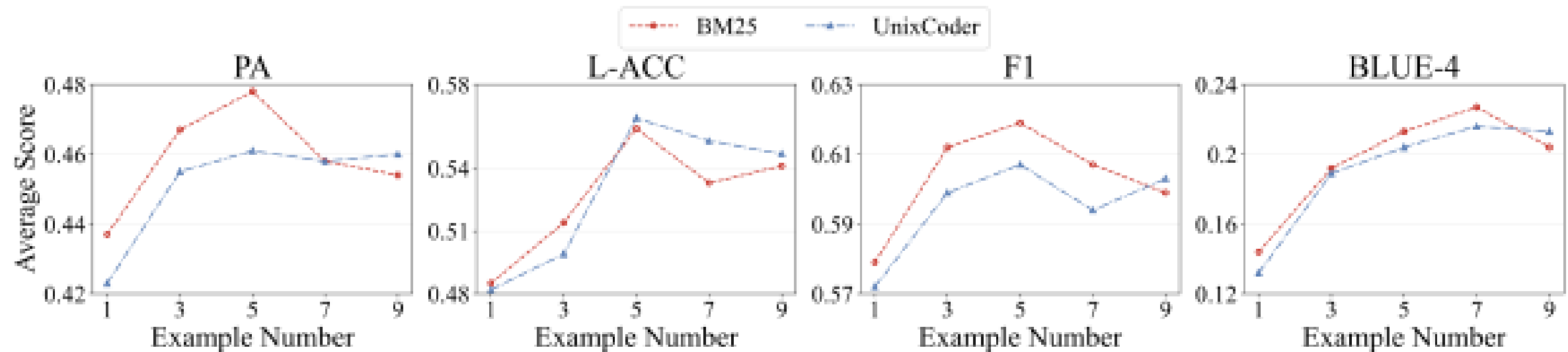


Fig. 8. The selected metrics of SCLOGGER with different numbers of examples and different sampling methods

## 例数の影響: 5個が最適

- BLUE-4のみ7が最大だが、それ以外はすべて5が最大

## 類似度計算手法: BM25 vs UnixCoder

- BM25が若干優位（テキスト類似度重視）

# 実用性の議論

---

## コスト削減:

- ソースコード全体を渡すよりも、プロンプトのサイズが削減されている
  - 必要最小限の文脈のみ抽出するため

## IDE統合:

- Eclipse などの統合開発環境に簡単に組み込める
- 既存コード補完ツールとの差別化
  - 最低限の文脈情報を考慮する点で差別化
- 様々なLLMとの互換性

# 貢献と成果

---

## 主な貢献:

1. 初の文脈化ログ生成手法の提案
2. 静的文脈をLLMに統合する新しいプロンプト構造
3. 包括的な評価による有効性の実証
4. オープンソース化による研究コミュニティへの貢献

## 技術的革新:

- メソッドレベルを超えた文脈の活用
- プロジェクト固有パターンの学習
- 段階的変数精緻化メカニズム

# 限界と今後の課題

---

## 現在の限界:

- Java言語に限定
- データリークの可能性（軽微）
- モデル選択の制約

## 今後の研究方向:

- 他のプログラミング言語への拡張
- より精密な静的解析手法の導入
- コード特化型LLMとの組み合わせ
- リアルタイム開発環境での実証研究

# 所感

---

かなり手が込んでいる研究

実は「SCLoggerによるログステートメントでログを出力したらどうなるか？」という話には触れられていない

ログが必要以上に生成され、ログが見つらくなっている可能性がある(仮説)

見やすいログを生成するような手法の提案が必要



# 限界1：限定的な静的スコープ

---

問題：logErrorメソッド単体では目的が不明

解決策：呼び出し関係を辿ることで「サービス停止エラー」の記録が目的と判明

- メソッド間の実行フローと利用可能変数の把握が重要

## 限界2：一貫性のないログスタイル

---

問題：プロジェクト固有のログパターンを学習できない

- 例：類似メソッドは全てdebugレベルで"Start doing..."パターンを使用

解決策：同一プロジェクト内の類似メソッドを参考例として活用

# 限界3：変数型情報の欠如

---

問題：変数のメンバ関数・属性情報が不明

- 例：`service.getServiceState()`、`service.getName()`の使用

解決策：詳細な型定義情報の提供